

MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

**A Data Structure  
For VLSI Synthesis  
And Verification**

*digital integrated systems cen*

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
AI	



## A Data Structure For VLSI Synthesis And Verification

David W. Knapp and Alice C. Parker

Digital Integrated Systems Center Report  
DISC/83-6

Department of Electrical Engineering-Systems  
University of Southern California  
Los Angeles, California 90089-0871

28 October 1983

**DTIC**  
**ELECTE**  
**S** FEB 15 1984 **D**  
**D**

This research was supported by Army Research Office Grant #DAAG29-80-k-0083, the Department of the Army, Ft. Monmouth, N. J., Contract #DAAK20-80-c-0278, the National Science Foundation, #MCS-8203485, and the International Business Machines Corporation, contract # S 956501 Q LX A B22.

Approved for public release; distribution unlimited.

The views, opinions, and/or findings contained in this report are those of the authors, and should not be construed as an official department of the army position, policy, or decision, unless so designated by other documentation.

**DISTRIBUTION STATEMENT A**  
 Approved for public release;  
 Distribution Unlimited

# Table of Contents

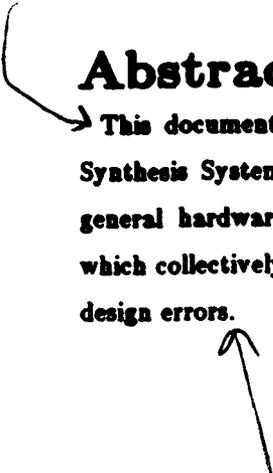
<b>Abstract</b>	<b>1</b>
<b>1. Introduction</b>	<b>3</b>
1.1. Motivation	3
1.1.1. Relationship to The USC Expert Synthesis System	4
1.1.2. The need for good data structures	4
1.1.3. Properties of the Design Data Structure: Requirements	5
1.1.3.1. The design representation and the design library	5
1.1.3.2. Completeness of the data structure	6
1.1.3.3. The man-machine interface	6
1.2. Features of the data structure	7
<b>2. The four hierarchies</b>	<b>9</b>
2.1. Fundamental concepts	9
2.1.1. Two approaches to design representation	9
2.2. Objects in the four hierarchies	11
2.2.1. Dataflow subspace	11
2.2.2. Sequencing	12
2.2.3. Structure subspace	13
2.2.4. Physical subspace	14
2.3. Interspace Bindings	14
2.4. Hierarchies in the data structure	15
2.4.1. The dataflow hierarchy	15
2.4.1.1. Primitive and composite objects in the dataflow subspace	15
2.4.1.2. Use of subscripts for nodes and values	16
2.4.1.3. Unfixed loops and their effect on subscripting	17
2.4.2. The structure hierarchy	19
2.4.3. The physical hierarchy	19
2.4.3.1. Bounding Boxes	20
2.4.3.2. Pins	20
2.4.3.3. The natural physical hierarchy	21
2.4.4. The sequencing hierarchy	21
2.4.4.1. The endpoints of a range	22
2.4.4.2. Links that denote hierarchical relationships	23
2.4.4.3. Predicates	24
2.4.4.4. Inheritance of predicates	26
2.4.5. On methods for dealing with large hierarchies	26
<b>3. A small example</b>	<b>29</b>
<b>4. Acknowledgements</b>	<b>33</b>
<b>Appendix I. syntactic definitions</b>	<b>35</b>
1.1. Syntax Conventions	35
1.2. Syntactic and Semantic Definitions	35

## List of Figures

<b>Figure 2-1:</b>	<b>Subscripting: Effect of Unfixed Loops</b>	<b>18</b>
<b>Figure 2-2:</b>	<b>Illustration of the Timing Hierarchy</b>	<b>22</b>
<b>Figure 3-1:</b>	<b>A simple example showing function, structure, timing and bindings.</b>	<b>30</b>
<b>Figure 3-2:</b>	<b>A more complex example showing structure, timing and bindings</b>	<b>31</b>

## Abstract

→ This document describes a VLSI design representation developed at USC as part of the USC Expert Synthesis System project. The data structure is implementation-independent and can be regarded as a general hardware design representation schema. It is characterized by four nonisomorphic hierarchies, which collectively describe the system under design. It is useful for the analytic detection of some kinds of design errors.



# 1. Introduction

Design automation (DA) technology has advanced to the point where almost all of the lower-level design tasks, and some of the higher-level tasks, can be automated. In such a situation, the implementation of a number of independent DA programs, each with its own design representation, can cause serious problems in the areas of data translation and consistency. As higher-level DA techniques are developed, these problems will get worse unless some effort is made to unify the representation of design information. At present, the manner in which design information is represented, coupled with the informal and implicit design rules usually embedded in synthesis software, make program-driven verification, incremental design, user interaction and backtracking difficult or impossible in real-world situations. The design data structure (DDS) described in this document is the result of an effort to eliminate some of these problems.

Unified data structures from the logic level down are becoming common; data structures which take into account behavior are more unusual. Nestor [11] links structure and behavior in the CMU DA system but doesn't completely solve the many-to-many linking problems. Davis [5] proposes a representation and Brown and Stefik [2] propose to represent both structure and behavior.

The above research tends to represent design as a single hierarchy. As a result, structure and behavior become intermingled in the representation, and there is redundant information contained across levels.

## 1.1. Motivation

The design data structure described in this document was designed to be the fundamental internal data structure of an expert system for digital hardware synthesis. In order to support general kinds of synthesis tasks, it has to represent:

- the functional behavior of a target machine,
- its structure,
- its control and timing, and
- its physical attributes.

Furthermore, it must represent the relations between these four classes of information. In order to formalize the synthesis process, these relations are explicit and can be thought of and mathematically treated as relations over sets of design information entities.

### **1.1.1. Relationship to The USC Expert Synthesis System**

An expert synthesis system (ESS) is currently under construction at USC [7] as part of the ADAM DA system. Its mechanisms for using design tools to manipulate design information are all unified into a hierarchy of knowledge domains, and it uses the unified design data structure throughout the entire design process.

The ESS embodies formal models of those parts of the design process that we do understand well, and supports the use of heuristic rules where formal models are unavailable or impractical. The use of explicit relations in the design representation facilitates the formal modelling process.

The primary objectives of the ESS are:

- To support interactive design
- To support incremental design
- To facilitate rapid adaptation to new methods and technologies
- To allow varying degrees of user interaction

These primary objectives lead us to secondary objectives, i.e. objectives that are a consequence of the primary objectives:

- Start with a system framework rather than a specific application
- Keep design data, rules, and code that implements rules separate
- Allow sharing of code and data between applications

Other expert-systems approaches to the problem of hardware synthesis have been reported in, e.g., [8], which attacks the problem of synthesis without backtracking and with a fixed order of task performance; [12], which addresses some issues of low-level electrical design, and [2], which describes the Palladio effort at Xerox PARC.

### **1.1.2. The need for good data structures**

We feel that the place to start in a large software system of this kind is with the data structures that the system will have to use. These data structures will fall into a number of classes, principal among them being representations of:

1. The target design
2. The design library of components
3. The system knowledge base
4. The system's current goals and activities

5. The history of the design process

6. Temporary "scratchpad" data structures

This document describes the target design data structure. Requirements and techniques for the system knowledge base, the system internal data, history, and scratchpad structures will be discussed in later documents.

### **1.1.3. Properties of the Design Data Structure: Requirements**

The design data structure should have the following properties:

1. The design data should be unified.
2. The representation should be directly executable.
3. It should facilitate analytic detection of design errors.
4. It should support the evolution of whole families of architectures.
5. It should provide a mechanism for translating between the data formats of different software tools, hence facilitating their use.
6. It should allow the design effort to be partitioned in a clean and well-structured way.
7. It should facilitate hardware synthesis under program control.
8. It should minimize the redundancy of the design data.

#### **1.1.3.1. The design representation and the design library**

Components in the design library are represented in the same way that the target design is represented; indeed, the target design could be regarded as an incomplete component in the design library. This way of regarding the target design would facilitate concurrent design of multiple components, which might be

1. Competing implementations of the same functional unit, one of which will be selected at some stage,
2. Implementations of several functional units intended to be used in a single target,
3. Implementations of unrelated components.

Furthermore, many of the management and control issues and strategies will appear in both the design library and the target representation.

It is, however, important to distinguish between a library of available components and the target design representation, because the operational requirements may be different. For example, an external vendor's component may be regarded as a "black box", if it is not a component of the system under design; its internal details need not be instantly available. On the other hand, the internal details of a non-primitive

component of the system under design should be readily available to the designer. Furthermore, in the case of a component that is only partially designed, an enormous amount of information must be instantly available.

In general, because the target component is operated upon a great deal in the course of its synthesis, access efficiency is far more important in the target structure than in the design library.

One way to create a well-structured, easily modified, global design representation might be to use a data base; clearly, the requirements for such a data base are different from those for a design library. A human user of a design library, and even to some extent a mechanical user of such a library, might find many disk access delays tolerable; but a synthesis program that depended on the data base as its basic data structure would perform poorly, even if it had a substantial "cache" data structure of its own. A design-state representation data structure, distinct from any design library, is described in this document.

### **1.1.3.2. Completeness of the data structure**

Another requirement of the target representation is that it be comprehensive. The implementor of a task-specific design program can afford to ignore any information that is not relevant to the task at hand, but during the overall design process a great deal of interrelated information is needed.

To avoid many of the translation and consistency issues that we referred to earlier, we opted for a single, combined representation, composed of four hierarchies which are linked to one another by special relations called *bindings*.

### **1.1.3.3. The man-machine interface**

A brief introduction to the man-machine interface will now be given, because the manner in which the expert system will be used has also constrained the data structure.

A user begins a design by entering initial design data. Small pieces of information can be added incrementally by the designer, and the synthesis programs can detect where there is missing information of a given type, and can (in some cases) fill it in. Thus, the design grows on top of the specification rather than being transformed from specification level to implementation level in a rigid way. We therefore need a data representation which can support partial designs, and an incremental design style.

At the same time the design dataflow behavior is undergoing top-down decomposition, a structure is being generated, and the sequencing information becomes increasingly detailed. However, design does not proceed uniformly top-down all the way to layout. An initial floorplan may be generated for the physical design in parallel with the logic design, but the detailed layout will probably proceed bottom up as macros are constructed from cells. This process will also be seen in the structural design insofar as library

structures are used. Thus a mixture of top-down and bottom up design strategies will be used, and the data structure must be designed to support any mixture of these strategies.

## **1.2. Features of the data structure**

We will now examine the details of a data structure that is intended to support very general kinds of synthesis tasks. It is characterized by:

- The use of four "orthogonal" hierarchies to represent different ways of looking at the design
- The use of explicit relationships between and within the hierarchies
- Uniform syntax of object definition and instantiation

This data structure is part of the foundation of the expert synthesis system, whose capabilities will, it is hoped, eventually cover the full range of digital design activities.

## 2. The four hierarchies

### 2.1. Fundamental concepts

This section deals with two central concepts of the data structure: *orthogonality* and *hierarchy*. Orthogonality is a property whereby one aspect, or *subspace*, of a design (e.g. its structure) can be varied independently of another (e.g. its behavior). We use the term orthogonality rather loosely here since we do not have strict mathematical independence between subspaces.

#### 2.1.1. Two approaches to design representation

In the conceptual methodology described in [6], a design is thought of as a locus in an  $n$ -dimensional discrete space; the positions of the points in the locus correspond to design characteristics. A speed-power tradeoff curve, for example, can be regarded as a projection of this space onto a plane: one can choose at what point a circuit will operate, and so establish an "operating point" that describes the circuit.

In the act of choosing a particular speed-power tradeoff we have not specified very much; e.g. we have said nothing about the logic diagram of the target design. The speed-power tradeoff is a projection of the design space onto a two-dimensional graph, as is a logic diagram, a timing diagram, or a dataflow graph.

These projections of a design are analogous to the multiple views used by a mechanical designer. Usually, three views of a mechanical part must be drawn: these are the front, top and side views. Because they are projections along orthogonal axes, a drawing of one of the views establishes only a minimal commitment in the other views. A drawing of the side view, for example, establishes only the height of the part as far as the front view is concerned, and the length as far as the top view is concerned. All other variables are left unfixed. In this way the designer can establish those properties he needs conveniently, because at every stage only a minimal number of decisions must be made.

The analogy in electronic design is to project from the design space along some axis or set of axes. This axis or set of axes should be chosen so that properties that are of interest are exposed, while uninteresting properties are hidden. The choice of axes has a great effect on the appropriateness of the projection for one purpose or another.

The axis chosen for pedagogical purposes<sup>1</sup> usually results in a projection that exposes such features as parallelism, functionality, and high-level structure. This is analogous to an isometric or perspective view of the mechanical part; it is very useful in illustrating to humans the most important properties of the system under study, and the interactions between its different kinds of properties are exposed. For

---

<sup>1</sup>The choice of axes is almost always informal and implicit.

example, a description of a processor that was purely structural (i.e. a schematic diagram), can be combined with information that is purely functional (i.e. the functions of the various blocks) to expose the inherent parallelism of an architecture. A hardware description language such as ISPS [1] is a good example of such a projection along an axis which is not orthogonal to either dataflow, structure, or control, and hence implicitly combines dataflow, structure, and control information into a single representation.

It is, however, by no means clear that such a choice of axis is the best for automatic design. The implicit combination of very different kinds of information means that a synthesis program that is intended to transform structures, for example, must start by taking apart the design description to separate the structural information that is really important from the structural information that is an incidental byproduct of the description. Indeed, because it is difficult to distinguish the incidental from the essential in such a description, optimizing transformations may be inadvertently precluded.

Another way to represent the design is to project the design space along axes chosen so that the projections form subspaces that are as close to<sup>2</sup> orthogonal as possible; i.e. so that information in one subspace has no direct bearing on information in another.

The set of subspaces are

- data flow: this space covers data dependencies and functional definitions.
- sequencing: this space covers conditional branching, timing, and sequences of events.
- structure: this space covers the electrical topology of a digital circuit.
- physical: this space covers the physical properties of electronic components.

The syntax and semantics of the four subspaces will be discussed in detail in Section 2.2.

The axes we chose were in part a result of an iterative process of definition, counterexample construction, and redefinition. A few of the examples we used are:

1. Two machines, identical except for their microprograms. Structurally they are almost identical, but their behavior can be radically different.
2. Two machines whose logic diagrams are identical but whose implementations are in TTL and NMOS, respectively. Their logical behavior is the same, but their timing and physical properties are different.
3. Two machines, one of which is a microprogrammed emulation of the other. Again, dataflow

---

<sup>2</sup>We know very little about the topography of electronic design spaces in general, so it is difficult to talk about "orthogonality" and "distance" very precisely.

behavior is the same, but timing, structure, and physical properties differ.

4. Two machines identical except for the bins from which their components were taken. The one can have a faster clock rate than the other.

The examples above represent cases where machines identical from one point of view are greatly different from another. We can take advantage of these "equivalences" in order to maximize the designer's freedom, because it is possible to explore one subspace without exploring the others.

A design description is therefore not represented as a single hierarchy of a functional layer built upon increasingly detailed layers of structure, with the bottom layer being physical design; rather it is represented as a number of nearly independent subspaces each having its own hierarchical structure.

Each of these subspaces can be thought of as a hierarchy of objects, but it is important to note that the hierarchies are not isomorphic<sup>3</sup>. This can be illustrated by the following example: a general-purpose computer that is implemented, at the highest level, as a memory and a computation unit. This is all very well as far as it goes: the memory possesses a behavior, and the computation unit possesses a behavior. It is clear, however, that the two computer functions at the highest level (instruction fetch and instruction execute), both require both memory and computation structures. Thus a top-down decomposition of behavior is very different from a structural decomposition. More examples can be given. Suppose that the processor has only one adder, which is used for both address indexing and ALU operations: it is possible to partition the address function and the ALU function into separate functions, but then both functions map into the same physical adder. In such a case two items in one hierarchy map onto one item in another hierarchy. This problem is worse when it is looked at in another way: suppose that the designer wants a Fast Fourier Transform function, and must distribute it among ten or fifteen physical boxes and data buses. A pathological case is a Matrix Multiply function that shares some but not all of the physical hardware used by the FFT function. Time-varying mappings can also exist, as in the case of the scoreboard scheduler of the CDC 6600. Under such circumstances, it seems like the wrong approach to try to fit all of the design information into a single hierarchy.

## 2.2. Objects in the four hierarchies

### 2.2.1. Dataflow subspace

The dataflow hierarchy has two object types: *nodes* and *values*. A node represents a function capable of transforming data values in some way; hence square-root, boolean NOT, and FFT would all be classified as nodes.

---

<sup>3</sup>This fact is possibly the most important immediate consequence of orthogonality.

A value is the result of function application; it is not the same as a variable. A value can be generated only once, and two values are identical only if:

- they are both constants, and the constants are equal, or
- the value names are synonyms<sup>4</sup>.

This node/value notation forms a single-assignment programming language that is useful for describing abstract data flow in terms of its inherent parallelism. It is not terribly useful for expressing the flow of control; such information is properly to be found in the sequencing subspace. The information to be found in the dataflow hierarchy concerns the detailed functional definitions of values and the data dependencies of values on other values. Nodes and values may be symbolically subscripted in order to represent iterative behavior (i.e. looping). This will be described in Section 2.4.1.2.

### 2.2.2. Sequencing

The sequencing hierarchy contains information about control and timing. Such things as conditional sequences of events, the times between events, and the ordering of events are represented.

The objects of the sequencing hierarchy are *points* and *ranges*. A point is regarded as being an infinitesimal time-span, at which some event can be said to have occurred. Between pairs of points fall the ranges; a range roughly corresponds to a state of the target machine. Ranges are directed; that is to say that they indicate the direction of the flow of time. A range may have either a known, an approximately known, or a wholly unknown length; and any two points may or may not be linked by a range. The graph of points and ranges is a directed acyclic graph, corresponding to a partial ordering of events.

Parallel and conditional sequences of events are represented by sets of ranges falling between special types of *fork* and *join* points. A pair of parallel sequences, for example, will be described as a pair of ranges with a common *and-fork* source point and a common *and-join* sink point; the *and* denotes that all branches will be traversed in parallel. A pair of conditional (i.e. exclusive) branches is represented by a pair of ranges connected at both ends by *or-fork* and *or-join* points; to each of these ranges is attached a predicate<sup>5</sup> describing the conditions under which they become the taken paths.

It should be noted that the decomposition of machine behavior into data flow and control flow is a little artificial; in particular, the execution of conditionals can cause a good deal of confusion. Consider two cases:

1. The function  $f(x)$  is calculated in two different ways depending on what part of its domain  $x$

---

<sup>4</sup>Synonymous value names normally arise in the case of subscripted values; e.g.  $X_i$  and  $X_j$  are synonymous when  $i = j$ .

<sup>5</sup>The predicates of the timing subspace are actually more complex than this suggests; they will be further discussed in section 2.4.4.3.

falls into.

2. The machine  $M$  undergoes radically different sequences of states depending on the value of a single bit *interrupt*.

While it is possible to express both of these situations in either an augmented control or an augmented dataflow graph, the DDS is designed from a perception that in practical fact these are different kinds of choices. Unfortunately, the separation of conditionals into control and functional domains is not a partition; some kinds of conditionals can be classified either way.

The special points  $\alpha$  and  $\omega$  are used in the representation of loops. The point  $\alpha$  is the initial point of a loop, and the point  $\omega$  is the point at which the loop recycles to  $\alpha$ ; for most purposes these two points are identical. The  $\alpha$  and  $\omega$  points are given symbolic subscripts. These subscripts are the ones used in distinguishing values and nodes in different iterations of the loop.

### 2.2.3. Structure subspace

The structure hierarchy also has two object types: *modules* and *carriers*. The structure subspace represents the topology of the design and is in most respects similar to a schematic or block diagram. Modules are interconnected by carriers to form a description of electrical topology; no unstructured text such as is found in schematics (e.g. labels, comments, and the names of draftspersons) is attached to this description.

A module is merely a building block whose behavior is defined by bindings to graphs in the dataflow and sequencing subspaces, and whose physical structure is defined by a binding to a graph in the physical subspace. This subspace can come in handy when trying to describe, for example, an array of NOR gates. If they form a two-level product-of-sums network, they could be described in their circuit-diagram form in the structural subspace. In the physical world, however, they might have several radically different implementations: an NMOS PLA, an array of TTL DIPs, or an array of ECL flat packs. Each of these alternatives has important physical and performance consequences.

An important feature of a carrier is that it is the only kind of structural item that a value can be bound to; i.e. it has some kind of storage or transfer function. A module can store data (as in a memory) only insofar as it has internal carriers capable of storing that data. No corresponding constraint on allocation of nodes to modules need exist, however, because of such examples as shifts and permutations that can take place in interconnect wires.

### 2.2.4. Physical subspace

In the physical subspace there are two primitive object types: *blocks* and *nets*. A block is a physical entity, for example a PC board or a DIP; a net is a wire used to interconnect blocks.

Blocks and nets have such inherently physical attributes as size, weight, and cost. They also can be given labels that denote technology (e.g. TTL or CMOS), implementation strategy (e.g. PLA, random logic), function (e.g. "contact pad" and "pullup") and electrical parameters such as capacitance, inductance, source impedance, and resistance. At this time, the list of all tag types is incomplete.

A block has explicitly denoted *pins*, which are the only allowed connection points. The ways in which pins are connected to one another via nets will be discussed in section 2.4.3.

### 2.3. Interspace Bindings

Relationships between objects of the four subspaces are made explicit by means of *bindings*. Bindings provide formal, well-defined links between the four subspaces. These bindings provide the ability to explicitly and unambiguously determine all of the relationships between the four hierarchies; all such information is to be found in them. There are four types of interspace bindings.

1. **value-carrier-range**: this binding type denotes the association of data values to structural carriers over time ranges.
2. **node-module-range**: this binding type denotes the association of dataflow nodes to structural modules over time ranges.
3. **module-block**: this establishes a correspondence between a structural module and a physical block.
4. **carrier-net**: this establishes a correspondence between a structural carrier and a physical net.

The first two interspace binding types are ternary relations; in general, they are many-to-many-to-many. This is because operations may be multiplexed in space or in time, and because values can be in many (reusable) places at many times.

The second two interspace binding types are binary because a mapping from (schematic) structure to (physical) layout is not a time-varying mapping.

Each binding can have an attached predicate; these predicates are used in the case of subscripted objects, and describe the ways in which the subscripts correspond to one another.

## 2.4. Hierarchies in the data structure

Each of the four subspaces discussed in the previous section is a hierarchy of objects. These hierarchies will now be discussed, first as general hierarchies of definitions, and then as individual structures.

### 2.4.1. The dataflow hierarchy

#### 2.4.1.1. Primitive and composite objects in the dataflow subspace

The dataflow hierarchy is composed, as described above, of nodes and values. A value is given a type; it may be of a primitive type, or a composite type. The primitive types are:

1. a Boolean
2. an unsigned integer, having as an attribute a finite range of values
3. a signed integer, also given a range of values.

To the ranges of each of these primitive types we append a special value, *bottom*. Bottom signifies an irrelevant or unknowable value; it corresponds roughly to the "don't care" of classical logic design. A special type "floating-point" was also considered, but the conclusion was reached that the user of the data structure would probably need to construct his or her own definitions of floating point numbers, and that the operators that would use such numbers would also have to be defined on a case-by-case basis; thus little is lost by not including them.

A composite value type is constructed from primitive and composite values by a process of instantiation; e.g. a complex floating point number, which is composed of two fields "real" and "imaginary", each of which is itself composed of two fields "mantissa" and "exponent". Note that the range of a composite number is derivable from the ranges of its components; hence no range need be specified for any but the primitive integers.

Such things as two's complement and binary-coded decimal numbers will also have to be defined in terms of composite values.

As there is no consistent and complete standard for the representation of characters as bit vectors, character data typing was also left to the user. In order to facilitate this and other kinds of enumerable representations, an enumerated type definition is also provided, whereby individual values of the enumerated type can be directly bound to integer values of an underlying bit vector.

A node definition in the dataflow subspace consists of

1. a type name, i.e. the name of the node type being defined
2. a list of the names of all interface values, with value type and input/output tag information

3. a list of all internal value instantiations, giving them names and specifying their types
4. a list of internal node instantiations, giving them names and specifying types
5. a list of internal bindings

The internal bindings are the means by which the node definition is made from an assortment of nodes and values into a graph. Each such binding is of the form

**<dataflow-internal-binding> ::= <tag> <value-name> <node-name> <value-name>**

where

- the tag denotes the type of the binding, in this case value-to-node
- the first value name is the name of a value instantiated within the current node type definition
- the node name is the name of a node instantiated within the current node
- the second value name is that of an interface value of the node whose name is given in the binding

For example, in a definition of a node called **FFT**, the primitive node type **multiply** might be instantiated a number of times, the name of the first instantiation being **multiply-instantiation-one**. The type definition of **multiply** has the interface values **multiplier**, **multiplcand**, and **product**, to which it becomes necessary to bind **FFT**'s internal values **x1**, **y1**, and **s1** respectively; thus the following three bindings are created:

1. **x1** is bound to **multiply-instantiation-one**'s interface value **multiplier**
2. **y1** is bound to **multiply-instantiation-one**'s interface value **multiplcand**
3. **s1** is bound to **multiply-instantiation-one**'s interface value **product**

#### 2.4.1.2. Use of subscripts for nodes and values

Looping traditionally involves the re-use of variable locations, but this cannot be done in a single-assignment environment such as we have chosen to use. The way we indicate the correspondence between the successive operations and values of a loop is by the use of subscripts, as in

```
for every 1 <= i <= 10 do
  x[i] := y[i] ** 2;
```

With such notation, it is simple to analyze the inherent parallelism of the problem and to denote the assignment of distinct values and operations<sup>6</sup> to hardware which may or may not be capable of exploiting the parallelism of the definition of **s**. Because this kind of looping is very common, nodes and values may have a subscript attached. This leads us to another problem, that of loops with no fixed number of traversals.

---

<sup>6</sup>Clearly the exponentiations of the loop above are distinct operations; one can easily imagine them being implemented in separate hardware units. We avoided subscripting the operator in the interest of clarity.

### 2.4.1.3. Unfixed loops and their effect on subscripting

A *fixed loop* is a loop that will always be executed a fixed number of times. The indices of such loops are easily handled with constant subscripts, as shown in the preceding examples. However, sometimes more general constructs are useful:

```
repeat
  programcount := 0;
  reset := false;
  repeat
    fetch( instruction[ programcount ] );
    programcount := programcount + 1;
  until reset;
forever;
```

An *unfixed loop* is a loop which has an iteration count which is undeterminable at specification time; there are two such loops nested in the example. To distinguish explicitly between the values and operators of successive traversals of an unfixed loop, we need to do two things:

1. set some fixed limit to the number of values we are willing to consider. For example, for a machine with a five-stage instruction pipeline we need to consider five instruction cycles; that is the time required for the longest instruction to be (in some sense) completed.
2. Establish an equivalence relation between sets of values and operators from different traversals, so that each and any can be talked of as members of a class.

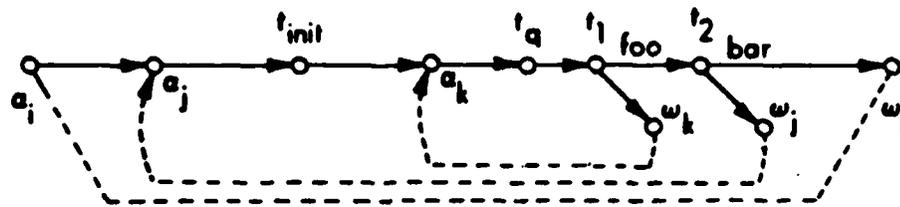
The formalism chosen to deal with this situation is a symbolic subscript. Any value or node may be fitted with a subscripting value or expression; such values as  $X_i$  and  $Y_{j-3}$ , as well as the node  $ADD_{k+2}$  can therefore be made explicit. It is important to note that the values used in subscripting expressions need have no physical counterparts in the target machine; not all loops have an explicit loop counter. Equivalence classes can then be defined as the user wishes, (e.g. as all those items whose subscripts are equal to zero in modulo-four numbering).

All subscript values therefore fall into a subrange of the positive integers; in the case of unfixed loops this subrange is finite but of unknown cardinality. Hence we might define a subscripting value in this manner:

```
var i : 0 .. I;
```

Note that in the case of a nested pair of unfixed loops, the innermost subscript  $i$  may well have many distinct values of  $I$  (final loop counts); these values are distinguished by a subscript on  $i$ , denoting the loop count of the outer loop in which the particular value of  $I$  happened to be the final loop count of the inner loop. While this kind of thing can be very annoying from a typographical point of view, it seems to be both essential and easy to deal with in a machine representation. An example of this situation is shown in Fig. 2-1.

The following two hierarchies (structural and physical) are very similar in their general syntax to the dataflow hierarchy; still, there are differences in interpretation and in content.



```

repeat
  repeat
    q = init
    repeat
      q = Xijk
    until foo
  until bar
forever
  
```

value	carrier	range	predicate
1	q	$a_i - a_j$	$i = 1$
1	q	$a_j - t_{init}$	$i = 1 \wedge j = 1$
init	q	$t_{init} - a_k$	$1 \leq i \leq j \wedge 1 \leq j \leq j_1$
init	q	$a_k - t_q$	$1 \leq i \leq j \wedge 1 \leq j \leq j_1 \wedge k = 1$
$X_{ijk}$	q	$t_q - t_1$	$1 \leq i \leq j \wedge 1 \leq j \leq j_1 \wedge 1 \leq k \leq k_j$
$X_{ijk}$	q	$t_1 - t_2$	$1 \leq i \leq j \wedge 1 \leq j \leq j_1 \wedge k = k_j$
$X_{ijk}$	q	$t_1 - w_k$	$1 \leq i \leq j \wedge 1 \leq j \leq j_1 \wedge 1 \leq k < k_j$
$X_{ijk-1}$	q	$a_k - t_q$	$1 \leq i \leq j \wedge 1 \leq j \leq j_1 \wedge 1 < k \leq k_j$
$X_{ij-1k}$	q	$a_j - t_{init}$	$1 \leq i \leq j \wedge 1 < j \leq j_1 \wedge k = k_{j-1}$
$X_{ijk}$	q	$t_2 - w_j$	$1 \leq i \leq j \wedge 1 \leq j < j_1 \wedge k = k_j$
$X_{ijk}$	q	$t_2 - w_i$	$1 \leq i \leq j \wedge 1 = j_1 \wedge k = k_{j_1}$
$X_{i-1jk}$	q	$a_i - a_j$	$1 < i \leq j \wedge 1 \leq j \leq j_{i-1} \wedge k = k_{j_{i-1}}$

Figure 2-1: Subscripting: Effect of Unfixed Loops

### 2.4.2. The structure hierarchy

The term "structure" in the present context is used to denote a set of design entities that correspond to schematic diagrams. Two types of objects are used: *modules* and *carriers*. Modules correspond roughly to the blocks of a schematic diagram, while carriers can have values bound to them and hence represent storage and interconnect.

The following rules are to be observed:

1. A module must have a defined type, which may be either primitive or composite.
2. A carrier must have a defined type, either primitive or composite.
3. In a module type definition, all internal and interface carriers are given their own names, which are unique in the context of the top level of the module definition. All of these carriers are carrier instances, and thus have associated types; each such carrier instantiation has an explicit type name associated with it.
4. A module instantiation is given its own unique name, and is tagged with its type name.
5. The header information of a module type definition contains the names and types of all interface carriers; each such carrier has associated with it a tag taking on one of the three values (input, output, bidirectional).
6. A module instantiation includes explicit bindings between the header (interface) carriers of its type definition, and the carriers to which the instantiation is connected.

### 2.4.3. The physical hierarchy

The physical hierarchy is composed of blocks and nets, as described in section 2.2.4. These blocks and nets obey the following rules:

1. Each block is either a primitive block or it is composed of blocks and nets.
2. Block and net instantiations have locally unique names, positions and transformations.
3. A block or net position may be any real-valued bound, set of bounds, or expression; i.e. it need not be a number. Furthermore, it may refer to the position expressions of other objects; hence, e.g., " $x > \text{block2}.x + 42\mu$ ", which says that the  $x$ -coordinate of the block is greater than the  $x$ -coordinate of another block *block2* by at least 42 microns.
4. A block or net instantiation made by means of a special subscripted instantiation represents a regular array of objects. Names in such an array are the array's base name and a fixed subscript; the array instantiation header includes position expressions and/or bounds that describe the detailed placement of each array element.
5. A block or net transformation consists of a rotation and/or a mirroring.
6. A net may only be connected to a block at a pin.
7. A net may be connected to a pin only if the pin is:

- a pin of the block inside of which the net is instantiated, or
- a pin of a block instantiated inside the definition. A net may connect pins across at most two levels of hierarchy<sup>7</sup>.

Because the physical hierarchy deals with concrete components, it is necessary to attach a good many attributes to blocks and nets; some attributes were discussed in section 2.2.4. The ways in which pins and bounding boxes are used is of some interest, and will now be discussed in detail.

### 2.4.3.1. Bounding Boxes

A bounding box is a rectangle that encloses all of the features of a block. This rectangle has two sides parallel to the *x*-axis. It is directly associated with the block type definition.

### 2.4.3.2. Pins

Pins are used to connect blocks to nets and to each other. They are the only interfaces a block may have. They are described in terms of:

1. A pin name, unique in the context of the block type definition to the electrical net in question (i.e. there may be many pins in a block that represent the same electrical point; they must all have the same pin name, and only they may have that pin name in the context of that block definition).
2. A position, consisting of either:
  - an *x-y* coordinate pair (intrachip and PCB coordinates), or
  - a pin number (chip carrier or other plug-in connector), or
  - a PCB edge connector number.
3. An optional perimeter segment descriptor.
4. A layer type tag (e.g. wire-wrap-pin, metal1, or PC-layer-4).

The pin name is locally unique to one electrical net; it may be re-used in another block definition, and it may be re-used in the current block definition if the re-use defines another connection point to the same electrical net.

The pin position denotes the point to which a connection net must be brought; its interpretation depends on the type tag of the block in question. The position may, for example, be an *x-y* coordinate pair in the case of a PCB or silicon-level block; but a for a block in a DIP the pin number should be used.

A pin with an *x-y* coordinate may be on the edge of the bounding box, or else inside it. If the pin is on

---

<sup>7</sup>It may, however, be connected *de facto* across any number of levels if at each level a new pin is defined and its connection attributes set equal to those of the lowest-level pin.

the edge, it may have a perimeter segment associated with it; in that case the pin may be thought of as having a width equal to the length of the perimeter segment, i.e. that a connection need only abut the segment at some point in order to exist. This can be useful in cases where an offset connection would cost area.

If, on the other hand, the pin is inside the bounding box, then it *must* have a perimeter segment associated with it. This segment denotes the part of the block which can be routed through on the layer of the pin. This "routable region" is defined as the smallest rectangle that includes both endpoints of the perimeter segment and the pin itself; the case where all three are collinear and bisect the block is specifically disallowed.

### 2.4.3.3. The natural physical hierarchy

There are several "natural" hierarchical levels in the physical domain:

- the network level: interconnected computer systems
- the mainframe level: sets of card cages and peripherals
- the card cage level: sets of printed circuit boards
- the PCB level: sets of chips and discrete components
- the chip level: sets of macrocells, gate arrays, or standard cells
- the silicon level: sets of transistors, diodes, and so on.

These are not modelled explicitly except insofar as blocks and nets can be tagged with technology and type labels, e.g. the block type "74LS02" which would be labelled as a low-power Schottky TTL DIP of 14 pins. Similarly a net might be labelled as implemented with #30 Kynar, or polysilicon: the "natural" levels are implicit in such labellings.

### 2.4.4. The sequencing hierarchy

In the sequencing hierarchy, a point represents an infinitesimal slice of time, during which an "event" can be said to take place. A range, therefore, is an infinite, ordered collection of points.

The basic structure of the timing hierarchy is shown in Fig. 2-2. A range, as shown in the figure, is either a composite range or a simple range. If it is a simple range, it could still be broken down later into simple ranges, thereby becoming a composite range; it can be infinitely divided.



```

case a of
  x: cobegin
      process1;
      process2
    coend;
  y: begin
      operation1;
      operation2
    end;
  z: . . .
  . . .
esac;

```

we can see that the entire case will generate a range that has a number of mutually exclusive alternatives, the active alternative being determined by the value of *a*. Furthermore, each alternative itself is represented by a range. The first of these, where  $a=x$ , is a parallel operation of two branches, while the second is a serial operation.

If we were to link the beginning of the case (i.e. its initial *or-fork* point) directly to the beginnings of its subranges, then the points beginning the subranges could be said to be the same point. That point would be a complex point with both exclusions and inclusions on its successor ranges, and rather complex groupings between the exclusions and the inclusions. In order to avoid these complications, dummy arcs representing a time range of duration zero must be used that establish the ordering explicitly in the case of a fork or a join point. A simple syntactic check will reveal any errors of dummy-arc omission.

#### 2.4.4.2. Links that denote hierarchical relationships

Levels in the timing hierarchy are tied together by links between points. Two linked points are identical in time; they have different names, but that is an artifact of the hierarchical description and not of the target machine.

Ranges can also be linked to one another, but no such identity holds. The main range of Fig. 2-2, called *instruction\_cycle*, is composed of, and linked to, five subranges: *x*, *fetch\_instruction*, *y*, *is\_jump*, *is\_at\_op*, and *z*. Each of those five subranges is itself linked to subranges. The range-subrange links are irreflexive, antisymmetric, and transitive.

The result of this strategy can be a great deal of redundancy, requiring a large storage space. The following arguments influenced our choice:

- It should be possible to get the length of a fully analyzed range (i.e. one whose subranges are of known duration) without recursively finding the path length from beginning to end of all its subranges, and then summing them. In other words, it should be possible to associate a length directly with a composite range.
- It should be possible to associate a predicate with a range without having to explicitly associate the predicate with all of the range's subranges.
- Because of the nonmonotonic nature of design, it should be possible to keep information about

the detailed structure of the subranges of a range, even though the structure of the composite range be completely known.

These three arguments are a reflection of two conflicting requirements: that it be possible to associate information directly with composite ranges for reasons of efficiency, and with subranges for reasons of completeness. The result is a tradeoff: both kinds of information can be represented, and a wise programmer will attempt to minimize redundancy by, e.g., refusing to store lengths in any but primitive ranges and ranges that are potentially under scrutiny. Such a policy would economize storage, and would tend to minimize the harmful effects of redundancy. It cannot be built into the data structure as it now stands, but must be enforced by the programmer.

### 2.4.4.3. Predicates

There are two kinds of sequencing predicates, and two ways in which a predicate can be inherited in the timing subspace.

The two kinds of predicate [9] are:

- *Synchronous* predicates, and
- *asynchronous* predicates.

A synchronous predicate is attached to each of the out-arcs of an exclusive branch point. Such a predicate indicates the conditions under which the out-arcs become active. For example, in the program

```
if a then foo else bar;
```

the branch *foo* would have the predicate *a* attached to it, indicating that it would only be traversed if *a* were true; similarly, *bar* would have the predicate *not-a*.

The asynchronous predicate is a slightly more difficult matter, and is needed to model such kinds of events as resets and timeouts. For example, the common 555 chip, which is a timer, can be used to implement a timeout; it consists of an analog timer and an RS latch with asynchronous clear. Upon being triggered, the analog timer is started and the latch is set; when the analog timer times out, the latch is reset. The *clear* input of the latch is taken from a pin of the 555; at any time after triggering, the latch can be reset by activating that pin, regardless of the state of the analog timer; and the act of clearing the latch discharges the timing capacitor.

How does one represent such a circuit? Conceivably it could be represented by, e.g.

```
repeat
  if trigger_edge then begin
    latch := set;
    while not (time or clear) do begin
      sample(time);
      sample(clear)
    end;
    latch := reset
  end
end
forever.
```

This representation has a fatal flaw in that it represents "busy waiting"; there is an implicit test of *clear* at every point in time, which results in a very large number of exclusion points, all with identical predicates, in the range that describes the timer's waiting time. Of itself this might be tolerable, but in the case of a more complicated sequence, e.g.

```

repeat
  programcount := 0;
  reset := false;
  repeat
    fetch(instruction(programcount));
    programcount := programcount + 1;
    execute
  until reset
forever.

```

This is clearly inadequate because *reset* is not tested until the end of the instruction cycle: if such a machine got jammed halfway through an instruction, this description would indicate there would be no way to reset it at all. This is appropriate for an interrupt, but for a genuinely asynchronous signal like *reset* it is inadequate.

One could test for *reset* in between every statement of the loop; but this is still inadequate, because some machines have block instructions. One could, in fact, push the *reset* test all the way down in the hierarchy, and insert a conditional (exclusion) point at the beginning of every primitive range: this is the extreme case of busy waiting, and even it will not work well unless all primitive ranges are as short as the shortest time interval of interest.

The solution we have come up with is the asynchronous predicate. It represents this situation in a manner like that of a domain quantifier in a function definition [3]:

$$f(x) = f_1(x), \quad x > X$$

$$f(x) = f_2(x), \quad x < X$$

In such a case  $f(x)$  is defined differently, depending on the value of  $x$ . Similarly, we can say that one set of ranges is the range of possibilities for a machine if *reset* is false, and another is the range of possibilities if *reset* is true. The predicate in this case is just *reset*: if it is false, the machine state is a normal-operation state, while if it is true, the state is a reset state.

There is another way to look at this formalism. If we specify the meaning of an asynchronous predicate to be an implicit exclusion fork at every point of every range that inherits that predicate, we have in effect created the kind of continuous monitoring we need. After all, a range is by definition infinitely divisible into subranges and points. Each of these implicit forks has two out-arcs: one to the next implicit or explicit point in the normal-operation timing graph, and one to the first point of the reset-operation timing graph.

The asynchronous predicate must therefore consist of:

1. A boolean expression that denotes the monitored value (in the case above the expression is just the value *reset*), and
2. A destination point to which control will pass if the expression ever becomes true.

#### 2.4.4.4. Inheritance of predicates

A predicate may be inherited in one of two ways. First, the predicate may be associated with a range's chronological predecessor, in which case it is inherited from the predecessor. This is the case with asynchronous predicates such as *reset*. It is also the case that information like that generated in the course of symbolic simulation [4] can be extracted by generating inheritances of synchronous predicates, but the information so extracted is not adequate by itself for symbolic simulation.

The other kind of inheritance is hierarchical inheritance, whereby the initial predicates of the first subrange of a range are the same as the initial predicate of the range itself. Some predicates, which are not invalidated during the course of subrange traversal, can safely be left attached to the superrange; all of its descendants can in this case inherit the predicates. In this case, the predicate is true also at the end of the superrange, i.e. at the beginning of its successor. Note, however, that it is possible for a predicate to be modified twice within a range, so that it is first true, then false, then true again: this is not the same situation and in some cases could cause problems.

#### 2.4.5. On methods for dealing with large hierarchies

In general, a hierarchical notation must express the relation between the elements of one level and the elements of the next. The DDS, which has interlevel relationships of a much more complex nature than "is-a-part-of", has been constructed with interlevel relationships that are as uniform as possible.

If we imagine a generalized hierarchy of interconnected elements as a graph problem, with composite nodes<sup>8</sup> that represent graphs and arcs that eventually, through some number of levels, connect to primitive nodes that are no longer divisible, we see that the chief notational pitfalls are:

1. the distinction between a node instantiation and its type definition,
2. the link between an instance of a node (be it primitive or composite) and its type definition,
3. the distinction between the structural arc that serves to interconnect instantiated nodes, and the arc that shows the relationship between the node instantiation and the node type definition (or in other words, the *connected-to* arc and the *composed-of* arc),
4. the fact that a *connected-to* arc may itself have a hierarchical structure, which implies that it

---

<sup>8</sup>NB. It is unfortunate that the graph-theoretic term *node* may easily be confused with the object type used in the behavior hierarchy; the reader is warned of this potential source of confusion.

may have *composed-of* arcs in its type definition and is a node itself in a hierarchy of *connected-to* arc type definitions,

5. the correspondence between an interface parameter (or formal parameter) of a node's type definition, and some particular *connected-to* arc on an instance of that node type,
6. the different identities of any two instances of a single type.

Henceforth we will assume that by default an *arc* is a *connected-to* arc; and that *composed-of* arcs will be explicitly denoted as such.

The base strategy for dealing with these problems in the DDS consists of the following guidelines:

1. An instantiated node must be of a defined type. That type may be either primitive or composite.
2. All arcs must also be of defined types, either primitive or composite.
3. In a node type definition, all internal and interface arcs are given their own names, which are unique in the context of the top level of the node definition; they are also tagged with their types.
4. In a node type definition, all interface arcs are explicitly denoted as such in the type definition header.
5. A node instantiation is given its own node name (unique in the context of the node type definition of which the instantiation is a part) and is also tagged with the name of its type.
6. A node instantiation includes explicit bindings between the formal parameter names of the type definition and the names of the arcs that are incident on the instantiated node.

This is only one set of conventions out of a number of possibilities; we felt that the establishment of a particular set of conventions was incidental to our entire research effort.



### 3. A small example

Figures 3-1 and 3-2 are partial examples of the way the data structure looks for a simple problem. At the top of Fig. 3-1 is a well-known formula; it is this formula that serves as an initial specification for a machine we will now "design".

Immediately below the formula we see a data flow graph. Nodes and values are labelled with their names, though some intermediate value names have been omitted for clarity. This data flow graph is a nonunique mapping from the equation.

Immediately below the data flow graph we see a tentative structure. It consists of a two-port register file  $R0-R5$ , an arithmetic unit  $A$  capable of performing all necessary operations, and an output register  $Y$ , all linked by bus structures.

Below the data flow graph is the timing graph. In this case, it is a very simple, word-serial implementation. Each range  $t_i-t_{i+1}$  corresponds to one arithmetic operation.

Below the timing graph is a partial list of bindings. On the left, value-carrier-range bindings are shown, while on the right node- module-range bindings are shown.

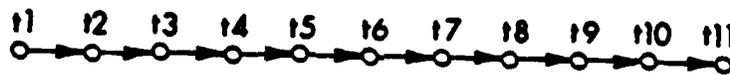
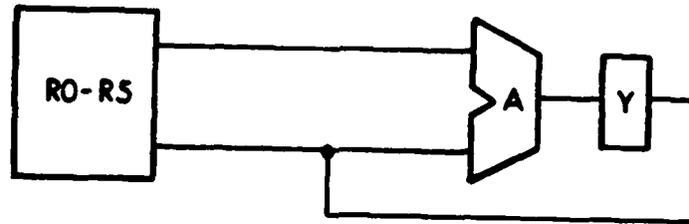
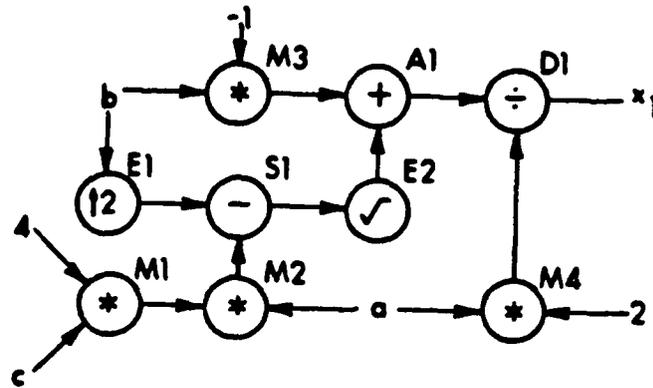
The first value binding shows that the value  $c$  is to be found in  $R0$  from  $t_1$  to  $t_4$ ; actually this would be three separate bindings, because in this diagram there is no hierarchy, and so there would be a binding for each range. The actual set of bindings have been omitted in the interest of clarity.

The first node binding shows that the multiplication node  $M1$  is performed by the arithmetic unit  $A$  during  $t_1-t_2$ .

Now let us suppose that the time span  $t_1-t_{11}$  was too long for us. Fig. 3-2 shows a possible solution: in this example the same function is implemented with more complex hardware. The single arithmetic unit  $A$  has been replaced by two units  $A$  and  $B$ ; each of which has been given dual input registers in the hope that the buses will be used more of the time. Part of the timing graph for this hardware is shown below the structure. The constraints upon the operations are far more complex, and bindings of values to the data buses are also shown, which was not the case in the previous example.

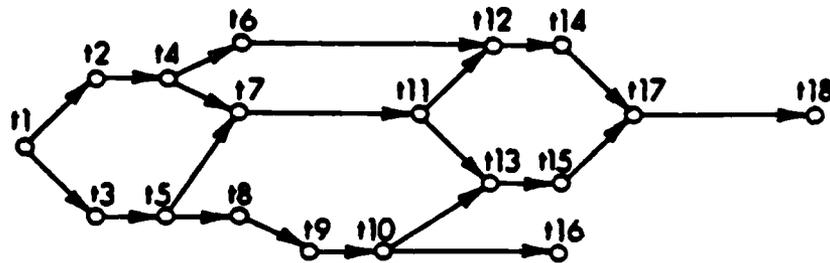
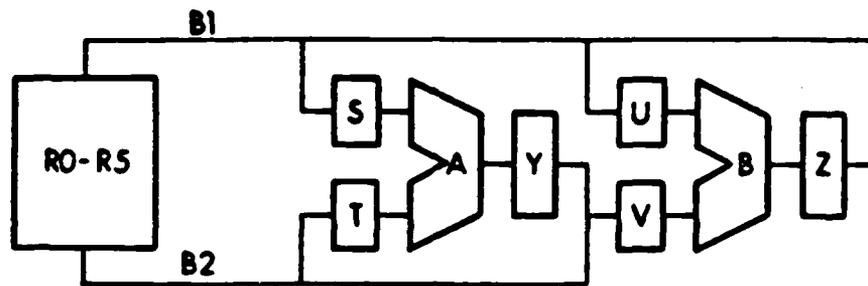
The first three value bindings shown at the lower left of Fig. 3-2 all describe what happens to the value  $c$ : it is to be found in  $R0$  initially and for some time after (that time being unknown at the "moment"); over  $t_2-t_3$  it is on the bus  $B1$ ; and over  $t_4-t_{11}$  it is in the register  $S$ . Because at  $t_4$  the value  $c$  is available in register  $S$  (and, incidentally, at  $t_5$  the value  $d$  is in  $T$ ), the operation  $M1$  can take place at any time afterward; that time is called  $t_7$  and the timing graph reflects those precedences. Note that setup and hold

$$x_1 = (-b + \sqrt{b^2 - 4ac}) / (2a)$$



Value	Carrier	Range	Node	Module	Range
c	R0	t1-t4	M1	A	t1-t2
4	R1	t1-t11	M2	A	t2-t3
V1	Y	t2-t3	E1	A	t3-t4
a	R2	t1-t11	S1	A	t4-t5
V2	Y	t3-t4	E2	A	t5-t6
V2	R0	t4-t7	M3	A	t6-t7
b	R3	t1-t11	A1	A	t7-t8

Figure 3-1: A simple example showing function, structure, timing and bindings.



Value	Carrier	Range	Node	Module	Range
c	R0	t1-	M1	A	t7 -t11
c	B1	t2-t6	E1	B	t10-t16
c	S	t4-t11	M2	A	t17-t18
4	R1	t1-		• • •	
4	B2	t3-t8			
4	T	t5-t11			
b	R2	t1-			
b	B2	t8-t10			
b	V	t9-t16			

Figure 3-2: A more complex example showing structure, timing and bindings

times are explicitly modelled by the intervals  $t_2-t_1$  and  $t_4-t_3$  respectively; all such times can be made explicit.

## 4. Acknowledgements

The authors would like to thank Mel Breuer, Bernie Cohen, John Granacki, Fadi Kurdahi, Mike McFarland, Nohbyung Park, and the entire VLSI expert systems group for their invaluable advice, comments, criticisms, and contributions.



# Appendix I

## syntactic definitions

### I.1. Syntax Conventions

The following conventions are observed for the syntactic definitions of objects in the lisp data structure.

1. 'item means that the item is a keyword or literal
2. <item> means that the item is a single instance of a generic item
3. N<item> means that the item is N instances of a generic item
4. {item} means that the item may be repeated zero or more times
5. <itema> ::= <itemb> denotes that itema is composed of itemb

This is a rather loose version of BNF.

### I.2. Syntactic and Semantic Definitions

1. <node-defn> ::= <node-type-name><nilist><nivlist><ninlist><niblist>

- Meaning: A node definition.
  - <node-type-name> is the name of the node type definition
  - <nilist> is the list of interface values to the node
  - <nivlist> is the list of the node's internal values
  - <ninlist> is the list of the node's internal nodes
  - <niblist> is the list of the node's internal bindings

- Notes: A node is a dataflow operator, e.g. ADD.

- Date: 9-83

- Originator: dwk

- Generation: 0

2. <value-defn> ::= <value-type-name><value-instance>{ <value-instance> }

- Meaning: a value type definition.
  - <value-type-name> is the name given to the defined value type
  - <value-instance> is a field in the defined type

- Notes: A value is not a variable. This is a single-assignment formalism.

- Date: 9-83
- Originator: dwk
- Generation: 0

3. **<module-defn> ::= <module-type-name><milist><miclist><mimlist><miblist>**

- Meaning: a module type definition
  - <module-type-name> is the name of the defined type
  - <milist> is a list of interface carriers
  - <miclist> is a list of internal carriers
  - <mimlist> is a list of internal modules
  - <miblist> is a list of internal bindings
- Notes: A module is a structural item, as one would see on a schematic.
- Date: 9-83
- Originator: dwk
- Generation: 0

4. **<carrier-defn> ::= <carrier-type-name><carrier-instance>{ <carrier-instance> }**

- Meaning: a carrier type definition
  - <carrier-type-name> is the name of the carrier type being defined
  - <carrier-instance> is a field instance
- Notes: A carrier is the same as a schematic "wire".
- Date: 9-83
- Originator: dwk
- Generation: 0

5. **<block-defn> ::= <block-type-name><bilist><binlist><billist><biblist>  
<shape><level> [fur.tag]**

- Meaning: a block type definition.
  - <block-type-name> is the name of the block type being defined
  - <bilist> is the list of interface pins
  - <binlist> is the list of internal nets

- `<billist>` is the list of internal blocks
- `<biblist>` is the list of internal bindings
- `<shape>` is the shape of the block
- `<level>` is the physical hierarchy level of the block
- `<funtag>` is a tag that denotes function
- Notes: A block represents a physical piece of hardware, e.g. a chip or a board.
- Date: 9-83
- Originator: dwk
- Generation: 0

6. `<net-defn> ::= <net-type-name><net-instance>{ <net-instance> }`

- Meaning: a net type definition.
  - `<net-type-name>` is the name of the type being defined
  - `<net-instance>` is the name of a field instance
- Notes: All nets must be of defined type. Th
- Date: 9-83
- Originator: dwk
- Generation: 0

7. `<range-defn> ::= <range-type-name><rulist><riplist><rirlist><riblist><duration>`

- Meaning: a range type definition.
  - `<range-type-name>` is the name of the range type being defined
  - `<rulist>` is a list of the range's endpoints
  - `<riplist>` is a list of internal points
  - `<rirlist>` is a list of internal ranges
  - `<riblist>` is a list of internal bindings
  - `<duration>` is a length of time which may be fixed or unfixed.
- Notes:
- Date: 9-83

- Originator: dwk
- Generation: 0

8. **<node-instance> ::= <node-name><node-type><subscript>**

- Meaning: a node instantiation
  - **<node-name>** is the name of the node being instantiated
  - **<node-type>** is the name of its type definition
  - **<subscript>** is the name of a list of subscripting values, which do not have to be instantiated in the target design
- Notes: An instance of a dataflow node.
- Date: 9-83
- Originator: dwk
- Generation: 1

9. **<value-instance> ::= <value-name><value-type><subscript><value-value>**

- Meaning: a value instantiation.
  - **<value-name>** is the name of the instantiation
  - **<value-type>** is the name of its type definition
  - **<subscript>** is the name of a subscripting value, which may not be instantiated in the target design
  - **<value-value>** is a temporary (simulation purposes) or permanent (constant) numeric value assigned to the value.
- Notes: A (singly assigned) dataflow value.
- Date: 9-83
- Originator: dwk
- Generation: 1

10. **<module-instance> ::= <module-name><module-type><subscript>**

- Meaning: a module instantiation.
  - **<module-name>** is the name of the instantiation
  - **<module-type>** is the name of the instantiated type
  - **<subscript>** is the name of a subscripting value, which may not be instantiated in the target design

- Notes: An instance of a structural module.

- Date: 9-83

- Originator: dwk

- Generation: 1

11. **<carrier-instance> ::= <carrier-name><carrier-type><subscript>**

- Meaning: a carrier instantiation.

- <carrier-name> is the name of the instantiation

- <carrier-type> is the name of the instantiation's type

- <subscript> is the name of a subscripting value, which may not be instantiated in the target design

- Notes: A carrier is equivalent to a "wire" on a schematic.

- Date: 9-83

- Originator: dwk

- Generation: 1

12. **<block-instance> ::= <block-name><block-type><transformation><subscript>**

- Meaning: a block instantiation.

- <block-name> is the name of the instantiated block

- <block-type> is the name of the block's type definition

- <transformation> is a CIF-style calling transformation (e.g. T x y MX)

- <subscript> is the name of a subscripting value, which may not be instantiated in the target design

- Notes: transformation will restrict all blocks to planar layout; must be extended for more general geometries

- Date: 9-83

- Originator: dwk

- Generation: 1

13. **<net-instance> ::= <net-name><net-type><subscript><transformation>**

- Meaning: a net instantiation.

- <net-name> is the name of the instantiation

- **<net-type>** is the name of the instantiation's type
- **<subscript>** is the name of a subscripting value, which may not be instantiated in the target design
- **<transformation>** is a CIF-style transformation (e.g. T x y R p q)
- **Notes:** A primitive net is a (rectangle, layer) pair, unless it is a wire.
- **Date:** 9-83
- **Originator:** dwk
- **Generation:** 1

14. **<point-instance> ::= <point-name><point-type><subscript>**

- **Meaning:** a point instantiation.
  - **<point-name>** the name of the point
  - **<point-type>** a tag denoting the type of the point
  - **<subscript>** is the name of a subscripting value, which may not be instantiated in the target design
- **Notes:** there is no way to define a point; the only types allowed are primitive points.
- **Date:** 9-83
- **Originator:** dwk
- **Generation:** 1

15. **<range-instance> ::= <range-name><range-type><range-predicate> <range-async-pred><subscript><duration>**

- **Meaning:** a range instantiation.
  - **<range-name>** the name of the range
  - **<range-type>** the type of the range
  - **<range-predicate>** is a predicate that describes the condition under which the range is actually traversed
  - **<range-async-pred>** is a predicate that describes the validity of a range
  - **<subscript>** is the name of a subscripting value, which may not be instantiated in the target design
  - **<duration>** is a length, which may be fixed or unfixed.

- **Notes:**

- Date: 9-83
- Originator: dwk
- Generation: 1

16. `<node-name> ::= <name>`

- Meaning: a node name.
- Notes: A name is not subscripted; an instance may be.
- Date: 9-83
- Originator: dwk
- Generation: 2

17. `<node-type> ::= <node-type-name> | <prim-node-type>`

- Meaning: a node has a type:
  - `<node-type-name>` the name of a node type
  - `<prim-node-type>` the name of a primitive node type
- Notes:
- Date: 9-83
- Originator: dwk
- Generation: 2

18. `<nilist> ::= { <value-instance> 'i' | 'o' | 'b' }`

- Meaning: a list of node interface values.
  - `<value-instance>` the type of the value in question
  - 'i' the value is purely input
  - 'o' the value is purely output
  - 'b' the value is structured and mixes input and output
- Notes: Mixed I/O values may not be needed.
- Date: 9-83
- Originator: dwk
- Generation: 2

19. `<nivlist> ::= { <value-instance> }`

- **Meaning:** a list of a node's internal values
- **Notes:**
- **Date:** 9-83
- **Originator:** dwk
- **Generation:** 2

20. **<ninlist> ::= { <node-instance> }**

- **Meaning:** a list of a node's internal nodes
- **Notes:**
- **Date:** 9-83
- **Originator:** dwk
- **Generation:** 2

21. **<niblist> ::= { 'vn <value-name> <node-name> <value-name> }**

- **Meaning:** a list of a node's internal bindings.
  - 'vn is a tag denoting the binding type
  - <value-name> is the name of the node's internal (or interface) value
  - <node-name> is the name of a subnode of the node
  - <value-name> is the name of the subnode's interface value
- **Notes:** The tag may be removed at a later date.
- **Date:** 9-83
- **Originator:** dwk
- **Generation:** 2

22. **<value-name> ::= <name>**

- **Meaning:** the name of a value.
- **Notes:** A name is not subscripted; an instance may be.
- **Date:** 9-83
- **Originator:** dwk
- **Generation:** 2

23. **<value-type> ::= <value-type-name> | <prim-value-type>**

- **Meaning: the type of a value.**
  - **<value-type-name>** is the name of a structured value definition
  - **<prim-value-type>** is the name of a primitive value type
- **Notes:**
- **Date: 9-83**
- **Originator: dwk**
- **Generation: 2**

24. **<module-name> ::= <name>**

- **Meaning: the name of a module**
- **Notes: A name is not subscripted; an instance may be.**
- **Date: 9-83**
- **Originator: dwk**
- **Generation: 2**

25. **<module-type> ::= <module-type-name> | <prim-module-type>**

- **Meaning: the type of a module.**
  - **<module-type-name>** the type of a structured module
  - **<prim-module-type>** a tag denoting a primitive module type
- **Notes:**
- **Date: 9-83**
- **Originator: dwk**
- **Generation: 2**

26. **<milist> ::= { <carrier-instance> 'i' | 'o' | 'b' }**

- **Meaning: a module's interface carrier list.**
  - **<carrier-instance>** is the instance of an interface carrier
  - **'i'** denotes that it is purely input
  - **'o'** denotes that it is purely output
  - **'b'** denotes that it is multiplexed in/out

- Notes: Multiplexing can be in time, as in a bidirectional data bus; or it can be in the form of a bundle of wires that carry signals in both directions.

- Date: 9-83

- Originator: dwk

- Generation: 2

27. `<miclist> ::= { <carrier-instance> }`

- Meaning: a list of a module's internal carriers

- Notes:

- Date: 9-83

- Originator: dwk

- Generation: 2

28. `<mimlist> ::= { <module-instance> }`

- Meaning: a list of a module's internal modules

- Notes:

- Date: 9-83

- Originator: dwk

- Generation: 2

29. `<miblist> ::= { 'cm <carrier-name> <module-name> <carrier-name> }`

- Meaning: a list of a module's internal bindings.

- 'cm is a tag denoting the type of the binding

- `<carrier-name>` is the name of one of the module's internal or interface carriers

- `<module-name>` is the name of a submodule

- `<carrier-name>` is the name of one of the submodule's interface carriers

- Notes: The tag may be deleted at a later date.

- Date: 9-83

- Originator: dwk

- Generation: 2

30. `<carrier-name> ::= <name>`

- **Meaning:** a carrier's name
- **Notes:** A name is not subscripted; an instance may be.
- **Date:** 9-83
- **Originator:** dwk
- **Generation:** 2

31. **<carrier-type> ::= <carrier-type-name> | <prim-carrier-type>**

- **Meaning:** the type of a carrier.
  - **<carrier-type-name>** the carrier is of a structured type
  - **<prim-carrier-type>** the carrier is of a primitive carrier type
- **Notes:**
- **Date:** 9-83
- **Originator:** dwk
- **Generation:** 2

32. **<block-name> ::= <name>**

- **Meaning:** the name of a block
- **Notes:** A name is not subscripted; an instance may be.
- **Date:** 9-83
- **Originator:** dwk
- **Generation:** 2

33. **<block-type> ::= <block-type-name> | <prim-block-type>**

- **Meaning:** the type of a block.
- **Notes:**
- **Date:** 9-83
- **Originator:** dwk
- **Generation:** 2

34. **<bilist> ::= { <net-name> 'i | 'o | '3st | 'oc <sourceZ> <location> <coordpair>  
<perseg> <layer> }**

- **Meaning:** a block's interface net list.

- o `<net-name>` name of an interface net
  - o 'i the net is input-only
  - o 'o the net is output-only
  - o '3st the net has a three-state driver in the given block
  - o 'oc the net has an open-collector driver
  - o `<sourceZ>` is a list of Thevenin equivalents
  - o `<location>` the coordinates of the net's contact point on the edge of the block, or a pin number (as in the case of a DIP)
  - o `<perseg>` is either null or is a perimeter segment
  - o `<layer>` the layer the net is on
- Notes: coordpair is not so good, e.g., for numbered pins on a DIP
  - Date: 9-83
  - Originator: dwk
  - Generation: 2
35. `<binlist> ::= { <net-instance> }`
- Meaning: a list of a block's internal nets
  - Notes:
  - Date: 9-83
  - Originator: dwk
  - Generation: 2
36. `<billist> ::= { <block-instance> }`
- Meaning: a list of a block's internal blocks.
  - Notes: Naming convention is awry. "Block Internal Block List" would interfere with "Block Internal Binding List".
  - Date: 9-83
  - Originator: dwk
  - Generation: 2
37. `<biblist> ::= { 'nb <net-name> <block-name> <pin-name> }`

- **Meaning:** a list of a block's internal bindings.
  - 'nb a tag denoting the binding type
  - <net-name> the name of an internal or interface net of the current block
  - <block-name> the name of a subblock
  - <pin-name> the name of an interface pin of the subblock
- **Notes:** The tag may be deleted at a later date.
- **Date:** 9-83
- **Originator:** dwk
- **Generation:** 2

38. <net-name> ::= <name>

- **Meaning:** the name of a net.
- **Notes:** A name is not subscripted; an instance may be.
- **Date:** 9-83
- **Originator:** dwk
- **Generation:** 2

39. <net-type> ::= <net-type-name> | <prim-net-type>

- **Meaning:** the type of a net.
  - <net-type-name> the net's type definition (structured net)
  - <prim-net-type> a primitive net type tag
- **Notes:**
- **Date:** 9-83
- **Originator:** dwk
- **Generation:** 2

40. <point-name> ::= - <name>

- **Meaning:** the name of a point.
- **Notes:** A name is not subscripted; an instance may be.
- **Date:** 9-83
- **Originator:** dwk

- Generation: 2

41. `<point-type> ::= 'si | 'of | 'oj | 'af | 'aj | 'al | 'om`

- Meaning: the type of a point.

- 'si simple point
- 'of or-fork point
- 'oj or-join
- 'af and-fork
- 'aj and-join
- 'al source point
- 'om sink point

- Notes: types 'al and 'om always have subscripts. Others may never; I do not know.

- Date: 9-83

- Originator: dwk

- Generation: 2

42. `<range-name> ::= <name>`

- Meaning: the name of a range.

- Notes: A name is not subscripted; an instance may be.

- Date: 9-83

- Originator: dwk

- Generation: 2

43. `<range-type> ::= <range-type-name> | <prim-range-type>`

- Meaning: the type of a range.

- `<range-type-name>` the range is defined elsewhere
- `<prim-range-type>` the range is primitive

- Notes:

- Date: 9-83

- Originator: dwk

- Generation: 2

44. **<rulist> ::= <point-name> <point-name>**

- **Meaning:** the list of a range's endpoints.
- **Notes:**
- **Date:** 9-83
- **Originator:** dwk
- **Generation:** 2

45. **<riplist> ::= { <point-instance> }**

- **Meaning:** list of points that fall inside a range (not including any points that belong to subranges)
- **Notes:** Endpoints of subranges can be bound equivalent to internal points.
- **Date:** 9-83
- **Originator:** dwk
- **Generation:** 2

46. **<rirlist> ::= { <range-instance> }**

- **Meaning:** all subranges of a range
- **Notes:**
- **Date:** 9-83
- **Originator:** dwk
- **Generation:** 2

47. **<riblist> ::= { 'pr <point-name> <range-name> <point-name> }**

- **Meaning:**
- **Notes:** The tag may be deleted at a later date.
- **Date:** 9-83
- **Originator:** dwk
- **Generation:** 2

48. **<name> ::= <char> { <char> | <digit> }**

- **Meaning:** the construction of a name.
- **Notes:**

- Date: 9-83
- Originator: dwk
- Generation: 3

49. `<shape> ::= <rect>`

- Meaning: the allowable shape for a block.
  - `<rect>` a rectangle
- Notes: May have to be added to later.
- Date: 9-83
- Originator: dwk
- Generation: 4

50. `<level> ::= 'Si | 'chip | 'pcb | 'card | 'cage | 'rack | 'frame | 'dist`

- Meaning: the allowed physical (intrinsic) hierarchy levels.
  - 'Si silicon
  - 'chip chip
  - 'pcb pc board
  - 'card pc card
  - 'cage card cage
  - 'rack relay rack
  - 'frame main frame
  - 'dist distributed system
- Notes: ha.
- Date: 9-83
- Originator: dwk
- Generation: 4

51. `<rect> ::= 'r <coordpair>`

- Meaning: a rectangle and its size.
- Notes: The rectangle is assumed to have corners at (0,0) and (coordpair).
- Date: 9-83

- Originator: dwk

- Generation: 4

52. `<coordpair> ::= 2<flonum>`

- Meaning: an x-y coordinate pair.

- Notes:

- Date: 9-83

- Originator: dwk

- Generation: 4

53. `<layer> ::= 'poly' | 'dif' | 'm1' | 'm2' | <wire> | <pplayer>`

- Meaning: the allowable set of routing layers

- 'poly polysilicon

- 'dif diffusion

- 'm1 metal 1

- 'm2 metal 2

- <wire> round wire with size and insulation tags

- <pplayer> pc trace with layer number

- Notes: May need to be augmented at a later date.

- Date: 9-83

- Originator: dwk

- Generation: 4

54. `<pplayer> ::= 'pcl' <fixnum>`

- Meaning: a printed circuit layer and its number.

- 'pcl tag denoting a pc layer

- <fixnum> layer number

- Notes:

- Date: 9-83

- Originator: dwk

- Generation: 4

55. `<prim-value-type> ::= 'bool | <unsigned> | <signed> | <enumerated>`

- Meaning: the allowable set of primitive value types.

- 'bool a boolean
- `<unsigned>` an unsigned integer
- `<signed>` a signed integer
- `<enumerated>` an enumerated type.

- Notes:

- Date: 9-83
- Originator: dwk
- Generation: 4

56. `<prim-carrier-type> ::= 'bit | <bus>`

- Meaning: the allowable types of primitive carrier

- 'bit one-bit line
- `<bus>` multibit bus

- Notes:

- Date: 9-83
- Originator: dwk
- Generation: 4

57. `<prim-net-type> ::= <wire> | <box>`

- Meaning: a primitive net is either a rectangle or it is a wire.

- Notes: clumsy.
- Date: 9-83
- Originator: dwk
- Generation: 4

58. `<prim-node-type> ::= <and> | <or> | <not> | <xor> | <nand> | <nor> | .....`

- Meaning: primitive node types. Still under construction.

- Notes:

- Date: 9-83

- Originator: dwk

- Generation: 4

59. `<prim-module-type> ::= <and> | <or> | <not> | <xor> | <nand> | <nor> | .....`

- Meaning: primitive module types. Still under construction...

- Notes:

- Date: 9-83

- Originator: dwk

- Generation: 4

60. `<prim-block-type> ::= <pulldown> | <pullup> | <pla> | <gate> | <register> | <arith> ...`

- Meaning: a set of primitive block types. Still under construction...

- Notes:

- Date: 9-83

- Originator: dwk

- Generation: 4

61. `<prim-range-type> ::= 'pr`

- Meaning:

- Notes: May want to expand later; don't change syntax.

- Date: 9-83

- Originator: dwk

- Generation: 4

62. `<transformation> ::= 4<transop>`

- Meaning: A transop is either null or it is a primitive transform operator.

- Notes: This is very like CIF [10].

- Date: 10-83

- Originator: dwk

- Generation: 5

63. `<unsigned> ::= 'usv <fixnum> <fixnum>`

- Meaning: fixnums denote range.

- Notes:
- Date: 10-83
- Originator: dwk
- Generation: 5

64. `<signed> ::= 'sv <fixnum><fixnum>`

- Meaning: fixnums denote range
- Notes:
- Date: 10-83
- Originator: dwk
- Generation: 5

65. `<enumerated> ::= 'env { <name> <fixnum> }`

- Meaning:
  - name is the name of an enumerated value
  - fixnum is the number to which it is bound
- Notes:
- Date: 10-83
- Originator: dwk
- Generation: 5

66. `<bus> ::= 'bus <fixnum>`

- Meaning: a bus and its width
- Notes:
- Date: 10-83
- Originator: dwk
- Generation: 5

67. `<range-predicate> ::= a well-formed formula`

- Meaning: a predicate that describes the conditions under which a particular branch of a conditional will be traversed.
- Notes:

- Date: 10-83
- Originator: dwk
- Generation: 5

68. **<range-async-pred>** ::= a boolean-valued expression. Syntax TBD. Also includes a **<target>** field; that is a list of target pointnames and predicates, which describes where control is passed to.

- Meaning: a predicate that describes the conditions under which a range is asynchronously stopped and control is passed to another point.

• Notes:

- Date: 10-83
- Originator: dwk
- Generation: 5

69. **<interspace-binding>** ::= **<vcr-binding>** | **<nmr-binding>** | **<mb-binding>** | **<cn-binding>**

- Meaning: one of four types of interspatial binding.

• Notes:

- Date: 10-83
- Originator: dwk
- Generation: 5

70. **<vcr-binding>** ::= **<name>** **<name>** **<name>** **<subscript-predicate>** **<fixity>**

- Meaning: the names are the names of the value, carrier, and range respectively. The subscript-predicate describes the validity of the binding for different values of the subscripts; and the fixity describes the level of rigidity of the binding itself.

• Notes:

- Date: 10-83
- Originator: dwk
- Generation: 5

71. **<nmr-binding>** ::= **<name>** **<name>** **<name>** **<subscript-predicate>** **<fixity>**

- Meaning: analogous to vcr-binding above.

• Notes:

- Date: 10-83

- Originator: dwk
- Generation: 5

72. **<mb-binding> ::= <name><name><fixity>**

- Meaning: the names are those of the module and the block; the fixity is that of the binding itself.
- Notes:
- Date: 10-83
- Originator: dwk
- Generation: 5

73. **<cn-binding> ::= <name><name><fixity>**

- Meaning: analogous to mb-binding above.
- Notes:
- Date: 10-83
- Originator: dwk
- Generation: 5

74. **<perseg> ::= 'null | <segment>**

- Meaning: in some cases a perimeter segment is meaningless, e.g. on a DIP.
- Notes:
- Date: 10-83
- Originator: dwk
- Generation: 6

75. **<segment> ::= <flonum><flonum>**

- Meaning: a segment of the perimeter of a rectangle. Calculated by starting at the lower-left corner, and measuring counterclockwise; the two flonums describe an interval.
- Notes: it is the untransformed rectangle!
- Date: 10-83
- Originator: dwk
- Generation: 6

76. **<value-value> ::= <number> | <name>**

- **Meaning:** either a fixed numeric value, or the name of an enumerated type value.
- **Notes:**
- **Date:** 10-83
- **Originator:** dwk
- **Generation:** 6

77. **<sourceZ> ::= { <source> }**

- **Meaning:** a list of Thevenin equivalents
- **Notes:**
- **Date:** 10-83
- **Originator:** dwk
- **Generation:** 6

78. **<source> ::= <boolean-constant><voltage><impedance>**

- **Meaning:** the boolean is either "T" or "F"; the voltage and impedance describe the Thevenin equivalent of the driver at that logic level.
- **Notes:** No, it is not very complete.
- **Date:** 10-83
- **Originator:** dwk
- **Generation:** 6

79. **<wire> ::= <wiretag> <flonum>**

- **Meaning:** a tag denoting the wire's size and insulation, and its length.
- **Notes:**
- **Date:** 10-83
- **Originator:** dwk
- **Generation:** 6

80. **<box> ::= <rectangle> <layer>**

- **Meaning:** it is a rectangle on a certain layer. "wire" is disallowed.
- **Notes:** have to add analog properties, e.g. R, C, L.

- Date: 10-83
- Originator: dwk
- Generation: 6

81. **<location> ::= <coordpair> | <fixnum> | 'null**

- Meaning: either a set of coordinates, a pin number, or a tag denoting no location at all.
- Notes:
- Date: 10-83
- Originator: dwk
- Generation: 6

82. **<duration> ::= { <duration-clause> }**

- Meaning: the length of time something will take or last.
- Notes:
- Date: 10-83
- Originator: dwk
- Generation: 6

83. **<duration-clause> ::= <fixity> <duration>**

- Meaning: the length of time something will take or last, and the "importance" of its taking or lasting that length of time.
- Notes: a conjunction of well-formed formulae, each clause of which has a <fixity> associated with it.
- Date: 10-83
- Originator: dwk
- Generation: 6

84. **<transop> ::= <translation-op> | <mirror-op> | <rotation-op> | <>**

- Meaning: it is either a translation, a rotation, or a mirroring.
- Notes:
- Date: 10-83
- Originator: dwk

- Generation: 6

85. **<duration> ::= <absolute-duration> | <relative-duration>**

- Meaning: a measure of an interval of time

- Notes:

- Date: 10-83

- Originator: dwk

- Generation: 7

86. **<absolute-duration> ::= a real-valued formula with no variables.**

- Meaning: an interval.

- Notes:

- Date: 10-83

- Originator: dwk

- Generation: 7

87. **<relative-duration> ::= a real-valued expression on some set of durations and/or other variables and predicates.**

- Meaning: a duration expressed in terms of other durations.

- Notes:

- Date: 10-83

- Originator: dwk

- Generation: 7

88. **<translation-op> ::= 'T' <flonum> <flonum>**

- Meaning: a translation to x-y coordinates

- Notes:

- Date: 10-83

- Originator: dwk

- Generation: 7

89. **<mirror-op> ::= 'MX' | 'MY'**

- Meaning: mirroring about either the y or x axis respectively

- Notes:
- Date: 10-83
- Originator: dwk
- Generation: 7

90.  $\langle \text{rotation-op} \rangle ::= 'R \langle \text{fixnum} \rangle \langle \text{fixnum} \rangle$

- Meaning: rotation so that the old x-axis lies along the vector (fixnum, fixnum).
- Notes:
- Date: 10-83
- Originator: dwk
- Generation: 7

91.  $\langle \text{fixity} \rangle ::= \langle \text{absolute-fixity} \rangle | \langle \text{relative-fixity} \rangle$

- Meaning: how firmly a binding or other fact is established. There are essentially three levels or types of fixity
  - Absolute or infinite fixity. This denotes a requirement given to the designer.
  - Relative fixity, falling in the real-valued interval [0,1]. Decisions made with relative fixities are design decisions which may be retracted. The special value zero represents an unfixed, i.e. freely variable, decision: such a decision has been made but can be retracted for any reason.
  - Unspecified fixity, denoted by a negative fixity. This represents a quantity or decision which we wish to represent as existing, but as having no value, i.e. as not having been made or specified.

- Notes:
- Date: 10-83
- Originator: dwk
- Generation: 8

## References

1. M. Barbacci, G. Barnes, R. Cattell, D. Siewiorek. *The Symbolic Manipulation of Computer Descriptions: The ISPS Computer Description Language*. Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., August, 1979.
2. H. Brown and M. Stefik. *Palladio: An Expert Assistant for Integrated Circuit Design*. Memo KB-VLSI-82-17 (working paper), Xerox PARC
3. Bernie Cohen. *Private Conversation*. unpublished.
4. Darringer, J. *The Application of Program Verification Techniques to Hardware Verification*. Proceedings of the 16th Design Automation Conference, ACM -SIGDA, IEEE Computer Society DATC, June, 1979, pp. 375-381.
5. R. Davis, H. Shrobe, W. Hamscher, K. Wieckert, M. Shirley, S. Polit. *Diagnosis Based on Description of Structure and Function*. Proceedings of the National Conference on AI, AAAI, 1982, pp. 137-142.
6. S. W. Director, A. C. Parker, D. P. Siewiorek, D. E. Thomas. *"A Design Methodology and Computer Aids for Digital VLSI Systems."* *IEEE Transactions on Circuits and Systems CAS-28* (July 1981), 634-645.
7. D. Knapp, J. Granacki, and A. Parker. *An Expert Synthesis System*. Proceedings of the ICCAD 1983 conference, September, 1983.
8. T. Kowalski and D. Thomas. *The VLSI Design Automation Assistant: Learning to Walk*. 1983 IEEE International Symposium on Circuits and Systems, IEEE, 1983, pp. 186-190.
9. Mike McFarland. *Private conversation*. unpublished.
10. Mead, C. and Conway, L. *Introduction to VLSI Systems*. Addison Wesley, Reading, Mass., 1979.
11. J.A. Nestor and D.E. Thomas. *Defining and Implementing a Multilevel Design Representation With Simulation Applications*. ACM/IEEE Nineteenth Design Automation Conference Proceedings, 1982, pp. 740-746.
12. R. Zippel. *An Expert System for VLSI Design*. 1983 IEEE International Symposium on Circuits and Systems, IEEE, 1983, pp. 191-193.

1941  
1942  
1943  
1944  
1945  
1946  
1947  
1948  
1949  
1950  
1951  
1952  
1953  
1954  
1955  
1956  
1957  
1958  
1959  
1960  
1961  
1962  
1963  
1964  
1965  
1966  
1967  
1968  
1969  
1970  
1971  
1972  
1973  
1974  
1975  
1976  
1977  
1978  
1979  
1980  
1981  
1982  
1983  
1984  
1985  
1986  
1987  
1988  
1989  
1990  
1991  
1992  
1993  
1994  
1995  
1996  
1997  
1998  
1999  
2000  
2001  
2002  
2003  
2004  
2005  
2006  
2007  
2008  
2009  
2010  
2011  
2012  
2013  
2014  
2015  
2016  
2017  
2018  
2019  
2020  
2021  
2022  
2023  
2024  
2025

END

FILMED

3-84

DTIC